Vercel Course

Flavio Copes

Updated August 3, 2026

Contents

Deploy and operate a Next.js application with Git and the Vercel CLI, Preview and Production environments, scoped configuration, domains, logs, and rollback.

What you'll learn: Deploy a Next.js application through Git and the CLI, configure separate environments and a domain, diagnose failures, and rehearse a safe rollback.

Platform foundations

Prepare, import, build, and verify a Next.js project on Vercel.

What Vercel deploys

Understand projects, deployments, framework builds, global delivery, and why Next.js is closely integrated without being locked to one host.

Vercel turns a source project into immutable deployments and routes traffic to the deployment assigned to an environment or domain.

The platform has its deepest integration with Next.js, which Vercel maintains, but it supports many frontend frameworks and JavaScript or TypeScript server workloads. A Next.js app can run elsewhere. Keep application code portable where practical and identify which behavior depends specifically on Vercel before adopting it.

Separate three things in your mental model. A project stores configuration and integrations. A deployment is one immutable build with a unique URL. A production domain is a movable alias that sends traffic to one deployment. Promoting or rolling back changes the alias; it does not rewrite either deployment.

Immutability stops at the deployment boundary. Databases, object storage, queues, email providers, and third-party APIs continue changing independently. Record those dependencies and their environments. A previous deployment can still be unsafe to restore if its schema assumptions or credentials no longer match the external state.

List the static assets, server-rendered routes, functions, scheduled work, storage, and external services your course project needs.

Prepare the Next.js project

Verify the application locally and define its build, runtime, environment, and version-control assumptions before involving a deployment platform.

A platform cannot make a broken production build reliable. Start from the Field Notes app built in the Next.js course or another small repository you control.

Run linting and the production build, remove machine-specific paths, document required environment variables, and ensure generated files and secrets are ignored appropriately. Commit the source to Git. Vercel can deploy without Git through its CLI, but a repository gives the course a repeatable review and release workflow.

```
npm install
npm run lint
npm run build
git status
```

The clean-clone test is stronger than ”it works on my machine.” It reveals undeclared files, globally installed tools, missing lockfiles, case-sensitive path mistakes, and environment values that exist only in one shell. Pin the Node.js and package-manager versions the project depends on, and make the production build the same command Vercel will run.

Document each environment variable by name, purpose, scope, and whether it is required during build or request handling. Do not put secret values in the document. Add a health route that checks

application startup without exposing credentials or performing a destructive dependency test, then define one real user path for the post-deployment smoke check.

Clone the repository into a temporary folder and build it using only the committed files plus documented local environment values.

Import a Git repository

Connect a repository, choose the correct scope and root directory, and review detected settings before starting the first build.

Importing through the dashboard connects the project to a supported Git provider and creates an initial deployment from the selected repository.

Choose the personal account or team that should own the project, then verify the framework preset, root directory, install command, build command, and output behavior. Automatic detection is useful evidence, not a reason to skip review. In a monorepo, the root directory determines which package Vercel builds.

Ownership affects billing, domains, secrets, tokens, and who can recover the project. Select the durable team scope rather than whichever account happens to be logged in. Confirm the production branch and Git permissions before enabling automatic deployments.

Write down the detected settings before the first build. If the import fails, compare the repository root, install output, runtime version, and required build-time variables with the local clean-clone result. Changing several dashboard fields at once destroys that evidence; change one assumption and redeploy.

Import Field Notes, pause before deploying, and compare every detected setting with package.json and the repository layout.

Inspect the first deployment

Read build output, deployment metadata, generated URLs, source commit, and runtime behavior instead of stopping when the dashboard says Ready.

Every deployment records what source was built, how it was built, and the unique URL for the resulting output.

Open the deployment and read the build logs, duration, framework result, assigned environment, commit, and domains. Visit the generated URL and test a direct dynamic route, a form, and the health endpoint. A green build proves compilation and packaging succeeded; it does not prove every runtime path or external dependency works.

Record the immutable deployment URL and ID, not only the domain alias. The alias can move later, while the unique URL preserves the exact artifact you inspected. Confirm that the source commit, environment, build settings, and expected environment-variable names match the release.

Use direct HTTP evidence as well as a browser. Check status, redirects, cache headers, and content type for a static path, dynamic path, mutation, missing resource, and health route. If runtime behavior fails after a green build, move to runtime logs for that deployment and request rather than rebuilding unchanged source.

Test the home page, one dynamic note URL, a missing note, and `/api/health` from the deployment URL. Record any difference from local behavior.

Check your understanding: platform foundations

Check the Vercel project model, framework support, Git imports, build detection, deployment URLs, and ownership boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Git deployment workflow

Review Preview deployments and release through a controlled Production workflow.

Separate Preview and Production

Understand the default environments and keep unreviewed branch work away from the user-facing production domains.

Vercel provides Local, Preview, and Production environments. Each deployment belongs to an environment and receives its own URL.

A push to the configured production branch normally creates a Production deployment. Other branches and pull requests create Preview deployments. The production domain is reassigned only when a Production deployment succeeds. Keep environment values separate because a preview should not write to a production database by accident.

A Preview deployment is a real deployed application, not a harmless screenshot. It can send email, mutate storage, call paid APIs, and expose unfinished routes. Give Preview separate credentials and disposable data, protect it when necessary, and make destructive integrations inert unless the test requires them.

The unique deployment URL identifies one build, while branch aliases can move as new commits arrive. Verify the commit and environment shown by the deployment before reviewing it. A successful Preview also does not guarantee Production will behave identically when domains, secrets, data, or plan limits differ.

Create a feature branch with a visible footer change, push it, and compare its preview URL with the current production URL.

Review a Preview deployment

Use a stable branch URL and an immutable commit URL to test the exact change under review and communicate useful evidence.

A branch can receive many deployments. The branch URL tracks its latest deployment, while a commit-specific URL identifies one exact build.

Open the preview from the pull request or dashboard, verify the source commit, and test the changed paths plus a small regression set. Share the immutable URL when a review comment must remain attached to one version. If protection is enabled, make sure intended reviewers can access it without making private work public.

Review the environment as well as the interface. Confirm the preview uses test data and preview-scoped credentials, then exercise any migration, webhook, scheduled task, or third-party side effect

the change touches. A visual approval does not prove those boundaries are safe.

Leave evidence that can be repeated: commit SHA, immutable URL, paths tested, expected results, and any known difference from Production. If another push replaces the branch alias, the review should still point to the artifact it approved. Promote only the exact deployment or source revision that passed.

Update the branch twice. Identify the branch URL and both immutable deployment URLs, then write a short review tied to the latest commit.

Promote a reviewed change

Release through the production branch or a controlled promotion workflow and verify the domain moved to the intended deployment.

The common Git workflow merges an approved pull request into the production branch, triggering a new Production deployment.

Wait for the deployment to succeed before considering the release complete. Open the production domain, verify its deployment ID or commit, and perform the high-value smoke checks. Teams with explicit promotion settings can promote a tested deployment, but the same principle applies: know which immutable deployment receives production traffic.

Capture the previous production deployment before moving traffic. After the change, query the production domain and confirm it resolves to the intended deployment, not merely that the Preview URL still works. Test one static path, one dynamic path, one write, and the health signal while watching errors and latency.

Promotion moves code and built configuration; it does not reproduce Preview's external data or undo a Production migration. Keep schema changes compatible with both old and new code during the rollback window. If a smoke check fails, use the recorded previous deployment to restore traffic, then reconcile the production branch before the next automatic release.

Merge the reviewed footer change, watch the Production deployment, and record the old and new deployment identifiers.

Recover through Git

Revert a bad source change, let the platform build a new known-good release, and preserve an auditable repository history.

When the fault is in source code, a Git revert is often the clearest recovery. It creates a new commit that undoes the bad change.

Reverting preserves evidence and keeps the repository aligned with production. Push the revert to the production branch, monitor its build, and smoke-test the result. Platform rollback is faster when seconds matter and is covered later, but follow it with a source correction so the next deploy cannot reintroduce the same bug.

A Git revert changes source, so it creates a fresh deployment using current project settings and environment variables. That can be an advantage over restoring an old artifact, but it also means the result is not byte-for-byte identical to the previous release. Verify the new deployment ID and repeat the production checks.

Source recovery does not reverse database migrations, sent messages, deleted objects, or external

API calls. Before reverting, decide whether those effects are backward compatible, need compensation, or make the old code unsafe. Avoid force-pushing the production branch: an explicit revert gives reviewers and operators an auditable explanation of the recovery.

Introduce a harmless visible regression on a practice branch, deploy it to Preview, revert the commit, and confirm the new preview restores the prior behavior.

Check your understanding: git deployment workflow

Check production branches, preview deployments, commit URLs, pull-request review, promotion, aliases, and rollback by changing source.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Configuration and domains

Control builds, scope environment values, protect secrets, and connect a domain.

Control build settings

Know when framework detection is enough and when root, install, build, output, Node.js, or ignored-build settings need an explicit override.

Project settings define how Vercel turns the repository into a deployment. Framework presets provide good defaults, but overrides can silently change the artifact.

Keep commands in package.json when they are part of the project contract. Use dashboard overrides only when the deployment context truly differs and document them. Pin important runtime versions in repository files rather than relying on a dashboard default that another environment will not share.

Review the whole build input: root directory, package manager and lockfile, install command, build command, output directory, runtime version, and required build-time variables. In a monorepo, an apparently correct command can still run against the wrong package. Generated output should come from the build, not from an untracked local directory.

Configuration can drift because repository files and dashboard settings have different review histories. Prefer repository-owned settings when possible, and keep a short record for every dashboard-only override. After changing one, create a Preview deployment and compare its build log and output with the previous deployment before touching Production.

Compare the dashboard build settings with package.json and repository version files. Remove any redundant override or document why it exists.

Scope environment variables

Give Development, Preview, and Production their own values and understand that changes affect only deployments created afterward.

Vercel environment variables can target Development, Preview, Production, and supported custom environments. Preview values can also vary by branch.

Use separate databases, API keys, and feature settings where an environment can cause side effects. A variable change does not rewrite an existing immutable deployment; create a new deployment to use it. Values are available to project members with sufficient access, so team permissions remain part of secret security.

Scope variables by effect. Preview should not send production email, charge real accounts, or write to the production database. Branch-specific Preview values are useful for integration tests, but document the fallback so an unconfigured branch cannot silently inherit a dangerous target.

Environment changes apply to new deployments. Old immutable URLs may still execute with the values embedded or attached when they were built, so rotating a credential also means invalidating the old credential at its provider. Redeploy each required environment, verify a safe diagnostic that reports only `configured: true`, and remove temporary values after the test.

Add `APP_ENV` with different Preview and Production values, redeploy both environments, and render the non-secret value in a diagnostic page.

Keep secrets server-side

Store sensitive configuration in Vercel while preventing framework public prefixes, logs, build output, and client responses from exposing it.

Encrypting an environment value at rest does not make it safe to send to the browser or print into a log.

Use secrets only in server code. In Next.js, a `NEXT_PUBLIC_` prefix intentionally makes a value public at build time. Avoid logging request headers, tokens, or full environment objects. Rotate a value if it appears in a deployment log or client bundle, then redeploy and invalidate the compromised credential at its provider.

Server-only is a data-flow rule, not merely a filename convention. A server component or route can still leak a secret through serialized props, an error message, a redirect, or a response body. Return the result of using a credential, never the credential itself, and review build output for values that may have been inlined.

Use a different credential per environment with only the permissions that application needs. Rotation is complete only after the new deployment works and the old credential is revoked at the provider. Consider rollback too: restoring an older deployment may restore code that expects a credential you intentionally invalidated.

Add a fake secret to Preview, read it only in a server route that returns a boolean such as `configured`, and inspect browser source and logs.

Connect a custom domain

Add a domain, configure the exact DNS records Vercel requests, verify ownership, and confirm HTTPS and canonical routing.

A custom domain is an alias that routes to a deployment environment. DNS must prove and direct that relationship before traffic can arrive.

Add the exact hostname in project settings and follow the displayed record instructions; apex and subdomain requirements can differ. Avoid copying values from an old tutorial. Wait for DNS propagation and certificate issuance, then test both HTTP and HTTPS plus `www` and apex behavior. Choose one canonical host and redirect the other deliberately.

Separate the control planes during diagnosis. The registrar owns delegation, the DNS provider publishes records, Vercel verifies the hostname and serves the certificate, and the project assigns the domain to a deployment environment. A browser error does not tell you which step failed.

Use `dig` or another DNS query to inspect the public record, then `curl -I` to inspect redirect and HTTPS behavior. Record the previous DNS values before changing them and lower TTL ahead of a planned migration when appropriate. Domain rollback can restore routing, but DNS caches take time, so keep the old deployment healthy until the change has settled.

Use a domain or disposable subdomain you control, add it to the project, configure its DNS, and inspect resolution before opening it in a browser.

Check your understanding: configuration and domains

Check build settings, environment scopes, redeployment, secret boundaries, custom domains, DNS verification, and HTTPS.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

CLI and observability

Link and deploy from the terminal, separate logs, and watch useful production signals.

Install, link, and pull configuration

Connect a local directory to the intended Vercel project and retrieve Development configuration without committing local project metadata or secrets blindly.

The Vercel CLI lets you inspect and deploy projects from the terminal. Linking records which account and project the directory represents.

Run the current CLI through your package manager or `npx`, authenticate, and use `vercel link`. Inspect the generated `.vercel` metadata and keep it ignored unless your workflow explicitly requires otherwise. Pull Development environment values into an ignored local file and review its names without printing secrets to shared output.

```
npx vercel@latest link
npx vercel@latest env pull .env.local
```

Treat linking as a production-relevant choice. Before pulling or deploying, inspect the current account, project ID, and project name in `.vercel/project.json`. A directory copied from another repository can otherwise publish to the wrong project while every local command appears successful.

`env pull` creates a local snapshot; it does not keep values synchronized. Pull again when configuration changes, keep the destination ignored, and use commands that inject values without printing them where possible. After linking, run `git status --ignored` once to prove both `.vercel` and the local environment file cannot be committed accidentally.

Link Field Notes, verify the account and project IDs, pull Development values, and check Git status before running the app.

Deploy from the CLI

Create a Preview deployment by default, promote only with an explicit production command, and verify the exact URL returned by each operation.

Running `vercel` or `vercel deploy` creates a Preview deployment by default. Adding `--prod` targets Production.

The CLI is useful for repositories without a supported Git integration, experiments, and controlled automation. It can also build locally before upload when needed. Do not add `--prod` by habit: preview the artifact, inspect it, and promote deliberately. A CLI deployment does not replace source control or documentation of what was shipped.

```
npx vercel@latest deploy
# verify the returned preview URL
npx vercel@latest deploy --prod
```

The CLI deploys the source state in the current directory, which may include changes that are not in Git. Check `git status`, record the commit and any intentional local difference, and verify the linked account and project before uploading. Automation should pin or deliberately update its CLI version rather than silently changing behavior mid-release.

Keep the immutable Preview URL and deployment ID returned by the command. Test that artifact before using an explicit Production operation, then verify the production domain points to the expected deployment. If the release fails, move traffic back to the recorded known-good deployment and commit or discard the local source difference so a later Git deployment cannot surprise you.

Change a visible build label, create a Preview deployment through the CLI, verify it, then decide whether to make the same source a Production deployment.

Separate build and runtime logs

Diagnose compilation and installation failures from build output, and request-time failures from runtime logs tied to a deployment and route.

Build logs exist while Vercel installs and compiles the application. Runtime logs exist when deployed code handles requests and emits output.

Start with the phase that failed. A missing dependency or type error belongs to the build. A database timeout after a successful build belongs to runtime. Filter by project, environment, deployment, time, route, and severity where available. Preserve request or trace identifiers so one user report can be followed across services.

Build logs explain one immutable artifact: installation, framework detection, compilation, and output generation. Runtime logs explain invocations after that artifact receives traffic. Activity or audit history explains configuration and deployment changes. Choosing the wrong stream wastes time and can make a rebuild look like a fix.

Include deployment ID, route template, status, duration, request ID, and a safe error class in structured runtime logs. Never log full headers, bodies, cookies, or environment objects. Retention and available filters depend on the current plan, so route important alerts and evidence to a durable system rather than assuming the dashboard is permanent storage.

Create one Preview build failure and one guarded runtime failure. Find each in the correct log view and write the smallest factual diagnosis.

Observe production behavior

Choose a small set of traffic, error, latency, and resource signals that reveal user impact without turning every dashboard tile into an alert.

Observability helps explain how the deployed system behaves over time. It is useful only when the signals lead to a decision or investigation.

Watch request volume, error rate, high-percentile latency, function duration, and the external dependencies most likely to fail. Web Analytics and performance products may add useful client evidence, but check current plan availability and privacy implications. Define alerts around sustained user impact, then link each alert to a short investigation and recovery procedure.

Measure from the user's path outward. An increase in 5xx responses may come from application code, a database, or an upstream API; deployment IDs and request IDs connect the symptom to evidence. High-percentile latency shows the slow experience hidden by an acceptable average.

Every alert needs an owner, a time window, and a first action. Alert on a sustained error ratio or latency breach rather than a single failure, and include the production deployment and affected route in the notification. After a release, compare the new version with its predecessor, then either continue observing or execute the recorded rollback instead of debugging indefinitely under user impact.

Create a one-page operations note with four signals, their expected range, one alert condition, and the first log or dependency to inspect.

Check your understanding: cli and observability

Check CLI linking, environment pulls, preview and production flags, build logs, runtime logs, request investigation, and alert-worthy signals.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Production operations

Own plan limits, access, rollback, verification, and the complete project lifecycle.

Understand plans and usage

Match the current plan terms and resource limits to the application instead of relying on old prices, screenshots, or the word free.

Plan names, included usage, commercial-use terms, and prices can change. Treat the current pricing and limits pages as the source of truth.

Identify who owns billing, which workloads consume build time, bandwidth, functions, image optimization, logs, and other metered products, and what happens at a limit. Set available spend or usage controls and alerts before traffic grows. A free deployment is not permission to ignore acceptable-use or commercial eligibility rules.

Map each meter to application behavior. A bot can increase bandwidth and function invocations; an image-heavy page can consume transformation usage; noisy branches can multiply build minutes

and retained deployments. This makes a spike diagnosable and suggests a safe response such as rate limiting, caching, disabling an optional feature, or blocking an abusive source.

Record the date and source for plan assumptions because limits and included usage change. Use the platform's current controls where available, but also define an application-level degradation path. A cost alert that arrives without an owner or action is only a notification.

Create a usage budget for Field Notes with an owner, the current plan, three relevant meters, warning thresholds, and an action if usage spikes.

Protect deployments and access

Apply least privilege to team membership, project access, preview URLs, tokens, integrations, and production changes.

A deployment URL can expose unfinished features or test data. A broad token or team role can change every project in its scope.

Require strong account authentication, use the smallest team role, scope automation tokens narrowly, rotate unused credentials, and review connected Git and marketplace integrations. Use deployment protection appropriate to the data and audience. Do not put real customer data into Preview merely because the preview itself is hard to guess.

Draw the access path for a release: Git provider, Vercel team, project, automation token, Preview viewer, domain administrator, and external data store. Remove access at the narrowest point and verify that recovery still has at least one accountable owner. A token used by one project should not be able to deploy every team project.

Deployment protection can cover Preview or generated deployment URLs without making the production domain private. Test each hostname and environment from a signed-out session. Protection also does not sanitize the connected data, so keep Preview credentials and datasets separate even when authentication is required.

Review project members, tokens, Git access, integrations, and preview visibility. Remove one unnecessary permission or document why each remaining one exists.

Roll back a deployment

Move production traffic to a known-good immutable deployment quickly, verify recovery, and reconcile the repository afterward.

A rollback reassigns production traffic to an earlier deployment. It is useful when the current release causes immediate user harm.

Choose a deployment you previously verified, use the current dashboard or CLI rollback workflow, and smoke-test the production domain after the alias moves. A rollback does not undo a database migration or external side effect automatically. Revert or fix the source and redeploy so repository history and the next production build agree with the recovered state.

```
# inspect current CLI syntax before an incident
npx vercel@latest rollback --help
```

Select the target from evidence: deployment ID, commit, environment, age, previous smoke result, and compatibility with current data and credentials. An old deployment can contain a revoked secret or code that cannot read the current schema. ”The one before this” is not always a safe

target.

After traffic moves, test the production domain rather than only the old deployment URL, watch error and latency signals, and record the time to recovery. The database, environment configuration, scheduled effects, and third-party calls remain as they were. Compensate or roll those back separately where possible, then revert or fix the production branch so the next automatic deployment preserves the recovered behavior.

Write and rehearse a rollback drill using harmless visible changes. Record the trigger, chosen deployment, verification, and source follow-up.

Run the production checklist

Ship Field Notes with verified ownership, configuration, domain, security, monitoring, recovery, and documentation rather than only a successful build.

Finish the course by treating deployment as an operating responsibility. The final artifact is the application plus the evidence needed to keep it reliable.

Verify the production commit, routes, form mutation, errors, environment scopes, domain and HTTPS, access, logs, alerts, usage controls, and rollback target. Document how to deploy, diagnose, recover, rotate secrets, and remove the project. If the project is temporary, deleting it and its credentials is part of the lifecycle.

Attach evidence to the release: immutable deployment ID, commit, build result, production-domain response, high-value smoke checks, migration state, current alert owner, and known-good rollback target. Record which external systems can still make a code rollback unsafe.

The runbook should be executable by someone who did not build the project. Give exact places to inspect, a decision point for rollback, and the steps that reconcile Git, data, and credentials afterward. Finish with decommissioning instructions for domains, tokens, environment variables, integrations, and external data so a retired project does not remain an unowned security or billing surface.

Deploy the final Field Notes release, complete the checklist, perform one rollback drill, and ask another person to follow the deployment and recovery notes.

Check your understanding: production operations

Check plan limits, usage ownership, deployment protection, rollback, source reconciliation, production checklists, and decommissioning.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/vercel/.